UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Simulating music from the latent space of a Variational Autoencoder

Rubén Martínez Sastre
Tutor: Eduardo César Garrido Merchán
Ponente: Daniel Hernández Lobato

Junio 2019

Simulating music from the latent space of a Variational Autoencoder

AUTOR: Rubén Martínez Sastre
TUTOR: Eduardo César Garrido Merchán

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2019

Resumen

Este Trabajo de Fin de Grado desarrolla la adaptación de una herramienta de generación de simulaciones musicales pseudoaleatoria con otra herramienta que integra técnicas de composición artificial mediante interpolaciones.

Para comenzar se describirá la idea principal de proyecto seguido de la motivación que hay tras este trabajo. A continuación, se expondrán varias aplicaciones actuales que manejan la creación de música.

De igual forma, se comentarán las principales librerías que usan dichas aplicaciones y los modelos que integran. Acto seguido, se listarán los objetivos del trabajo, así como las hipótesis que criticaremos en el apartado de conclusiones. También se definirá el alcance del proyecto y las restricciones que se han dado durante el proceso del mismo.

Más adelante se definirá el flujo del programa y se desarrollará la forma de integración que tiene cada proceso. Se introducirán los modelos usados y su aplicación directa en el trabajo. Seguido de esto, se definirán para cada proceso los ficheros, almacén de datos, librerías, tecnologías usadas y versión de las mismas.

Para afianzar las bases del proyecto, se describirán los experimentos realizados y los resultados obtenidos.

Por último, se referencian los objetivos anteriormente mencionados y comentado el grado de satisfacción de los mismos, dificultades encontradas y lo aprendido. Se realizará el mismo proceso para las hipótesis lanzadas. Por último, se comentan las ideas de trabajo futuro.

Palabras clave

Melodía, interpolación, software, librería, expresión, realismo, suavidad, bpm, tempo.

Abstract

This End of Grade Work develops the adaptation of a tool for generating pseudo-random musical simulations with another tool that integrates artificial composition techniques through interpolations.

To begin with, the main idea of the project will be described, followed by the motivation behind this work. Next, several current applications that handle the creation of music will be exposed.

Likewise, the main libraries that use these applications and the models that they integrate will be discussed. Next, the objectives of the work will be listed, as well as the hypotheses that we will criticize in the conclusions section. It will also define the scope of the project and the restrictions that have been given during the project process.

Later on, the flow of the program will be defined and the form of integration that each process has will be developed. The models used and their direct application in the work will be introduced. Following this, the files, data warehouse, libraries, technologies used and their version will be defined for each process.

In order to strengthen the bases of the project, the experiments carried out and the results obtained will be described.

Finally, the previously mentioned objectives are referenced and the degree of satisfaction of the same, difficulties encountered and what has been learnt are commented upon. The same process will be carried out for the hypotheses launched. Finally, ideas for future work are discussed.

Keywords

Melody, interpolation, software, library, expression, realism, softness, bpm, tempo.

Agradecimientos

Si por mi fuera, gran parte de esta memoria trataría de agradecer a cada uno de los que han puesto su granito de arena y empeño, en que yo pudiera continuar realizando este proyecto.

A mi familia, por haberme apoyado en cada momento, siempre preocupándose de mí. No es mentira que la familia es lo primero, y menos cuando sabes que en todo momento estarán ahí, para todo, en lo bueno y en lo malo.

A mis amigos, por sacrificar su tiempo en hacer que no me explotara la cabeza. Sois los mejores amigos que uno puede tener, y me siento muy orgulloso de que seáis mi segunda familia.

A Alba Murillo, la compañera ideal de mi vida. Ha sabido soportarme en cada momento de incertidumbre, apoyando de forma increíble cada idea que de mi cabeza salía. Me has motivado, me has dado ánimos, me has dado vida. Te quiero más.

A Alberto Prudencio, por haber sido tan atento y servicial, por enseñarme el funcionamiento de su fantástico programa.

A Eduardo Garrido, por ser una fuente inspiradora de buenas ideas y contestar al segundo y con un detalle que asusta a cada ruego y pregunta que le he lanzado.

Y, por último, y no por ello menos importante, a aquellos que en su día supieron quererme y cuidarme, y que sé que lo siguen haciendo allá donde estén. Siempre han estado y siempre estarán.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE DIAGRAMAS

1 Introducción

1.1 Descripción y motivación

Dentro del mundo de la música existen múltiples paradigmas que a día sigue siendo un quebradero de cabeza para muchos. La frustración de luchar por algo y nunca alcanzarlo es uno de los muchos motivos por los que algunos compositores acaban abandonando. Quizás sus ideas no son del todo malas, pero quizás necesitan de una pequeña ayuda que les haga ver las carencias de lo ya creado, o lo que pueden llegar a ver. ¿Y si fuera posible crear una herramienta capaz de modificar ligeramente los patrones de una melodía, sin que la estructura principal, cohesión y equilibrio se viera afectados?

Esta memoria de TFG trata la integración de un TFG anterior (Alberto Prudencio de Dueñas) en el que se desarrollaba un simulador de música, para aprovechar su potencial y crear una herramienta capaz de generar interpolaciones de una melodía a otra. Este proceso seguirá una serie de pasos que serán definidos a lo largo de esta memoria y que concluiremos con una discusión de los experimentos realizados, así como una reflexión final del proyecto.

Existen varios problemas a resolver entre los que se encuentran carencias de conceptos, entendimiento de códigos ajenos y el amplio desarrollo de la herramienta de interpolación. Esta será descrita como una aplicación capaz de interpretar melodías autogeneradas por el *Artificial Composer* desarrollado por Alberto, y autogenerar por sí mismo otra melodía que surgirá de un espacio latente. Se defenderá, por lo tanto, una forma de interpolar ambas melodías que puedan crear una transición suave de una melodía a otra. Finalmente, a partir de una función de evaluación con unos criterios específicos, se calificará cada una de las melodías generadas.

Este proyecto nace de una idea, un *hobby* y una motivación que las une. ¿Es posible que un compositor se quede sin ideas? Francamente, si el compositor es muy bueno, esta idea se cuestiona. No obstante, puede existir una herramienta que le ayude a ser aún más creativo. De la afición a la música y el estudio del grado de informática, nace esta idea.

Bien es cierto que es muy difícil que una máquina iguale conceptualmente a las brillantes ideas de un gran compositor, pero se debe tener en cuenta el aprendizaje exponencial que estas están tomando los últimos años. Y bien es cierto que cada vez son más los que ponen su granito de arena apoyando a este tipo de proyectos, ya sea una empresa desarrolladora de *software* como es *Google*, o un modesto grupo de pseudoingenieros que simplemente quieren aportar sus ideas, que, aunque modestas, son brillantes e inspiradoras para otros.

1.2 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Introducción: introducción del proyecto en la que se describe el problema a resolver al igual que la motivación que se encuentra detrás de esta idea.

- Estado del arte: se exponen varias aplicaciones actuales que manejan la creación de música. De igual forma, se comentarán las principales librerías que usan dichas aplicaciones y los modelos que integran.

- Definición del proyecto: se listan los objetivos del trabajo, así como las hipótesis que criticaremos en el apartado de conclusiones. También se define el alcance del proyecto y las restricciones que se han dado durante el proceso del mismo.

- Diseño del proyecto: se define el flujo del programa y se desarrolla la forma de integración que tiene cada proceso. Se introducirán los modelos usados y su aplicación directa en el trabajo.

- Implementación: se define para cada proceso los ficheros, almacén de datos, librerías, tecnologías usadas y versión de las mismas.

- Experimentos: se describen los experimentos realizados y los resultados obtenidos.

- Conclusiones y trabajo futuro: se referencian los objetivos anteriormente mencionados y comentado el grado de satisfacción de los mismos, dificultades encontradas y lo aprendido. Se realizará el mismo proceso para las hipótesis lanzadas. Por último, se comentan las ideas de trabajo futuro.

2 Estado del arte

En este capítulo se presentan los diferentes caminos que la generación de música está tomando a nivel de desarrollo y que envuelven a este proyecto. Para poder tratar este tema de un modo más directo sin divagaciones, vamos a hablar de las diferentes aplicaciones que se están desarrollando desde el año 2018 y que, de un modo u otro, han logrado inspirar este proyecto.

La mayoría de estas aplicaciones han sido desarrolladas por *Google*. Es evidente que detrás de estos proyectos hay un trabajo inmenso con un equipo altamente cualificado y con una serie de recursos que se alejan mucho de un trabajo de fin de grado. No obstante, y sin tampoco desmerecer este humilde trabajo, a continuación, se exponen dichas aplicaciones, expuestas de forma incremental según su complejidad.

2.1 Melody Mixer

Un ejemplo claro de estas aplicaciones es *Melody Mixer*, desarrollado en el laboratorio creativo de *Google* y que demuestra, desde un perfil más simplificado que sus "competidores", el alto rendimiento de sus capacidades. *Melody Mixer* permite generar fácilmente interpolaciones entre bucles de melodía cortos.

La aplicación está principalmente desarrollada a través de cuatro grandes librerías, como son:

- ***P5.js* [1]:** librería que ayuda a que la codificación gráfica sea más sencilla. *P5* nos sirve para crear gráficos interactivos. Utilizando la metáfora original de un cuaderno de bocetos de software, *P5.js* tiene un conjunto completo de funciones de dibujo. Para esto, *P5.js* tiene bibliotecas de complementos que facilitan la interacción con otros objetos *HTML5*.

  La aplicación se apoya en esta librería para crear de forma sencilla un par de cuadros donde se mostrarán las melodías principales, las cuales usaremos para interpolar. Si arrastramos cualquiera de estos dos cuadros hacia un lado, se desplegarán tantos cuadros de melodía intermedios como interpolaciones se hayan creado. Estos se crean con un degradado de colores para un aspecto más llamativo de la aplicación.

- ***Tone.js* [2]:** marco de audio web para crear música interactiva en el navegador. La arquitectura de *Tone.js* pretende ser familiar tanto para los músicos como para los programadores de audio que buscan crear aplicaciones de audio basadas en la web.

  Tone brinda a la aplicación la capacidad de manejar sonidos como si de variables se tratara. De este modo podemos construir un amplio abanico de sonidos que, en conjunto, funcionarán como una melodía única. La demo web nos ofrece un único instrumento como es el piano, aunque como comentaremos en un apartado posterior, se pueden hacer múltiples adaptaciones para incluir desde otra familia de instrumentos como puede ser el viento metal, hasta cambiar totalmente de frente e incorporar percusión.

- ***TensorFlow.js* [3]:** biblioteca de inteligencia artificial acelerada por hardware para la web. Esencialmente, esto significa que podemos escribir comandos de *javascript* que se ejecutarán directamente en la *GPU*. No trabajaremos directamente con *TensorFlow.js*, pero es la herramienta subyacente que permite a MusicVAE.js [4], de la cual hablaremos a continuación, ejecutar un modelo de aprendizaje automático en tiempo real en el navegador.

- ***MusicVAE.js*:** la nueva biblioteca de *Magenta* [5] que nos permite combinar melodías y también probar / generar nuevas melodías. La biblioteca también funciona con ritmos de batería, pero esta versión web utiliza melodías para la demostración.

2.1.1 MusicVAE

Prueba y error. Así es como los grandes pintores, escritores y músicos han llegado a difundir con gran talante sus obras de arte. El probar, equivocarse, rectificar; todo forma parte de un proceso creativo que tiene como efecto un trabajo final sorprendentemente bueno. Si alguna vez un músico o compositor ha requerido de una herramienta que le otorgara ideas o tan sólo ligeras modificaciones, de cara a un trabajo digno de su grandeza, sin duda querría usar *MusicVAE*.

Como ya introdujimos en el apartado anterior, *MusicVAE* es un modelo de aprendizaje automático que nos permite crear diferentes patrones que podemos combinar y explorar partituras musicales. Utiliza una de las grandes premisas de la informática como son los modelos espaciales latentes. El objetivo de este modelo no es más que ser capaz de representar la variación de un conjunto de datos de alta dimensión mediante un código de menor dimensión, lo que facilita la exploración y manipulación de las características o atributos de los datos a tratar. Aunque está más enfocado al ámbito de la recomendación, hablaremos de su alta capacidad para, por ejemplo, distinguir entre diferentes géneros musicales.

2.1.2 Espacios latentes

Cualquier obra musical está compuesta por cientos de notas. Entonces, ¿es fácil generar música aleatoriamente? O, mejor dicho, buena música. Consideremos un espacio en el que nos encontramos con un piano de 88 notas en 9 octavas distintas. Si definimos cada una de las notas como un evento distinto y añadimos dos más, uno para para la intensidad de la nota y su duración (lo definimos como el mismo evento para simplificar) y otro para el momento en el que el músico descansa, obtenemos un total de 90 eventos distintos. Imaginemos un escenario en el que partimos de un 4/4, donde, por consiguiente, tendríamos un total de 90^{32} secuencias posibles. Si extendemos eso a 16 compases obtenemos 90^{256} secuencias posibles, lo cual excede por mucho el número de átomos del mundo. ¿Cuántas de estas secuencias pueden calificarse como una melodía coherente?

Los modelos espaciales latentes son capaces de aprender las características fundamentales de un conjunto de datos de entrenamiento y, por lo tanto, pueden excluir estas posibilidades no convencionales.

Además de excluir los ejemplos no realistas, los espacios latentes pueden representar la variación de datos reales en un espacio de dimensión inferior. Esto significa que también pueden reconstruir ejemplos reales con alta precisión. Además, al comprimir el espacio del conjunto de datos, los modelos de espacio latente tienden a organizarlo en función de las cualidades fundamentales, que agrupan ejemplos similares y establecen la variación a lo largo de los vectores definidos por estas cualidades.

Las propiedades principales de un espacio latente se pueden definir como:

- **Expresión:** cualquier dato real puede asignarse a algún punto del espacio latente y volver al reconstruir el dato real a partir del valor comprimido.

- **Realismo:** cualquier punto en este espacio representa un ejemplo realista, es decir, cualquier dato de entrenamiento o test debe ser un dato real.

- **Suavidad:** los puntos cercanos entre si tienden a tener cualidades similares. Si formamos un *cluster* de datos estos pueden tener los mismos *bpm* o simplemente compartir un género musical.

Dentro del uso que le podemos dar a los espacios latentes, cabe destacar la interpolación de puntos para construir una transición entre dos dibujos, como es el caso de la aplicación *SketchRNN* [6] desarrollada por Google.

2.1.3 SketchRNN

La propiedad realismo de la que hablábamos en el **punto 2.1.2** le permite a *SketchRNN* muestrear al azar nuevos ejemplos que no han sido introducidos por el usuario, pero que son similares al conjunto de datos real. Podemos seleccionar aleatoriamente un punto y decodificarlo para demostrarlo.

Figura 2-1: Ejemplo de *SketchRNN*

También podemos usar la estructura del espacio latente para realizar transformaciones semánticamente significativas, como con "restricciones latentes" (*Latent constraints* [7]).

2.1.4 Autoencoder

Hay muchos modelos diferentes capaces de aprender representaciones en un espacio latente, cada uno de ellos respeta las tres propiedades que deseamos: expresión, realismo y suavidad.

Uno de los modelos es el conocido *autoencoder* (AE). Un AE sigue los siguientes pasos:

1. Crea un espacio latente con los datos a aprender (*Encoder*).

2. Codifica o comprime cada ejemplo en un vector de números (código latente, o *Z*).

3. Decodifica, descomprime o reproduce el mismo ejemplo de ese vector de números (*Decoder*).

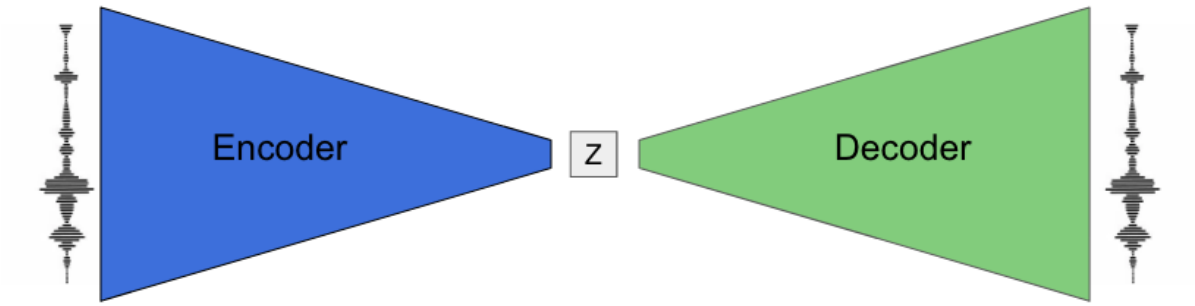

Figura 2-2: *Encoder, Z* y *Decoder* de un AE

Un componente a tener en cuenta es el cuello de botella introducido al hacer que el vector tenga menos dimensiones que los datos en sí, lo que obliga al modelo a aprender un esquema de compresión. En el proceso, el AE determina aquellos atributos o características que son comunes entre el conjunto de datos. *NSynth* [8] es un ejemplo de AE que ha aprendido un espacio latente de notas musicales.

Una de las limitaciones con este tipo de codificadores es lo que se denomina un "agujero" en su espacio latente. Esto significa que, si cogemos un vector aleatorio, puede que no resulte realista, si, por ejemplo, este se aleja mucho de nuestra nube de muestras reales. En el caso de *NSynth*, es capaz de reconstruir e interpolar, pero carece de la propiedad de realismo de la que hablábamos en el **apartado 2.1.2**, y, por lo tanto, pierde credibilidad a la hora de reproducir aleatoriamente un punto debido a estos "agujeros".

Otra forma de imponer nuestro cuello de botella que evada el problema anterior es mediante el uso de lo que denominamos un *variational loss*. En lugar de limitar las dimensiones de los vectores, podemos optar por producir códigos latentes con una estructura predefinida, como, por ejemplo, con muestras de una distribución normal multivariada. De esta forma nos aseguramos de que lo que vayamos a reconstruir produzca algo más realista.

2.1.5 Estructura a largo plazo

Generar estructuras coherentes a largo plazo es una de las carencias de los modelos mencionados anteriormente. Quizás *SketchRNN* nos ofrecía una forma de codificar una estructura a largo plazo para producir bocetos completos. Sin embargo, para lograr un resultado similar en secuencias musicales los suficientemente extensas como para que se asemejen a una canción (con muchas más variables que un dibujo), encontramos que no podemos confiar en las arquitecturas ya presentadas. Para resolver dicha encrucijada, basta con generar una estructura a largo plazo a partir de códigos latentes individuales.

Partimos de la idea principal del AE, pero en vez de inicializar el decodificador *RNN* de la nota directamente, insertamos el código en un *RNN* aparte llamado *conductor* que genera una nueva codificación para, en este caso, cada compás de la salida. La nota que sale del *RNN* genera cada uno de los compases de forma totalmente independiente.

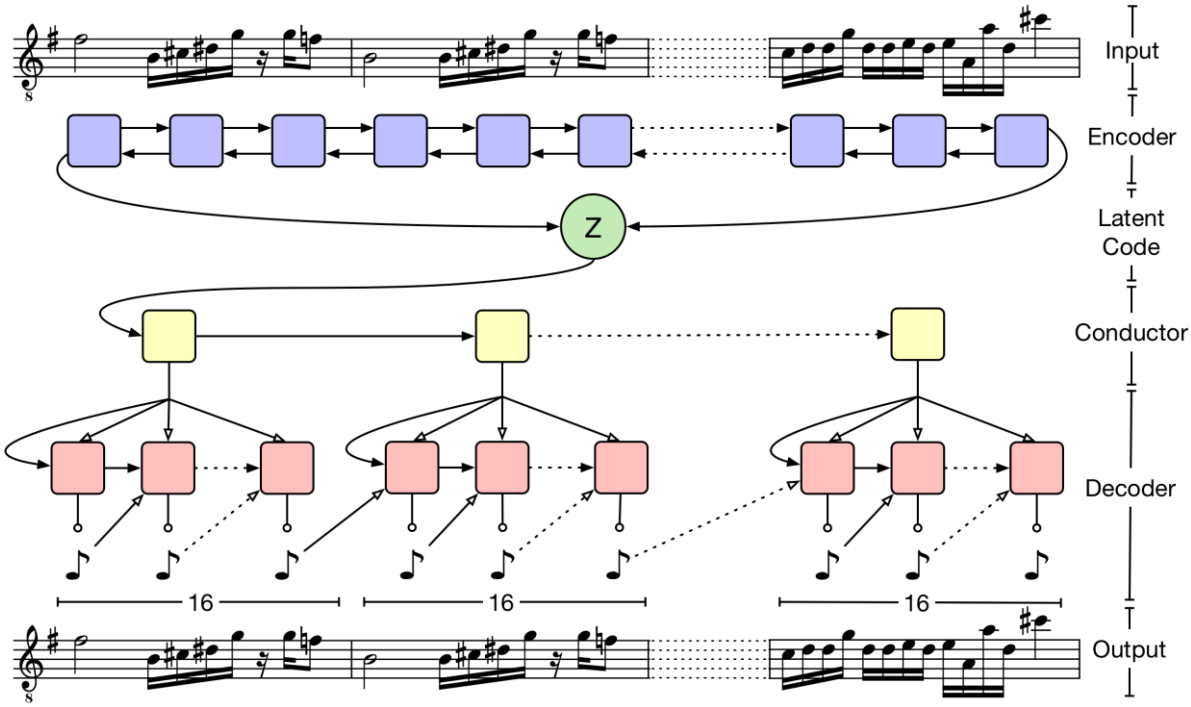

Figura 2-3: Pasos de *MusicVAE*

Como veremos en el apartado 5 (Integración, pruebas y resultados), encontramos que esta dependencia adicional marca la diferencia en las interpolaciones producidas. Dado que el modelo no podía simplemente recurrir a la regresión automática en el decodificador de notas para optimizar la pérdida durante el entrenamiento, esto hace que gane una mayor robustez en el código latente para reconstruir cada una de las nuevas secuencias.

Usando esta arquitectura, somos capaces de reconstruir e interpolar sin los problemas de antes, pero adicionalmente lo podemos hacer para melodías más largas.

2.2 Beat Blender

Otra aplicación es *Beat Blender* también desarrollada por *Creative Lab* de *Google*. La idea es similar a la de *Melody Mixer*, interpolar melodías. Pero esta aplicación va un poco más allá e implementa una serie de funciones, aparentemente básicas, pero que hacen de *Beat Blender* una aplicación con un alto potencial.

En este caso nos encontramos con la posibilidad de interpolar no dos, sino 4 melodías al mismo tiempo. Estableciendo una matriz donde las esquinas son las entradas, conseguimos reproducir un número de interpolaciones igual a 11x11 (absteniéndonos de las cuatro entradas de las que partimos).

Estas cuatro esquinas de las que hablamos pueden tomar valores melódicos predeterminados, como son un ritmo de rock, break, pop o samba. No obstante, si queremos introducir melodías de nuestra propia cosecha, la aplicación nos permite editar las entradas a partir de un mapeo de 4/4 con nueve sonidos diferentes: bombo, caja, platillo, timbal… De esta forma conseguimos un nuevo ritmo o ritmos para poder interpolarlos.

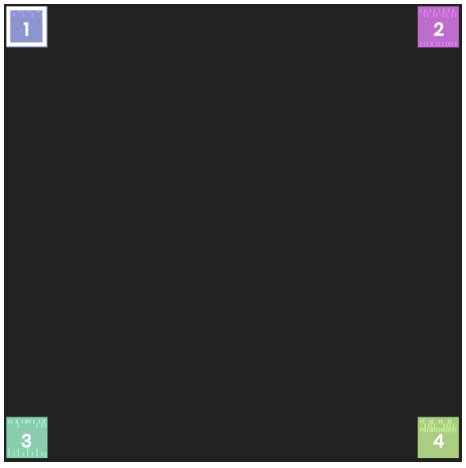

Figura 2-4: *Inputs* de la matriz

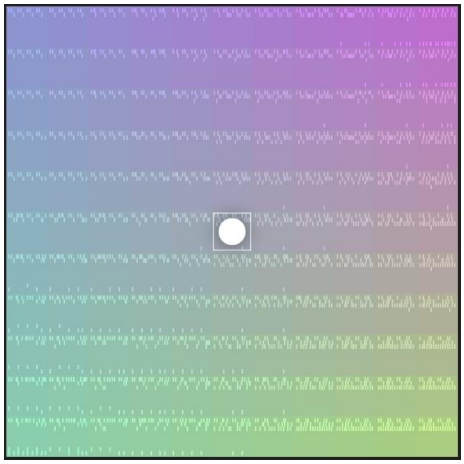

Figura 2-5: Matriz de interpolaciones

Es fácil percatarse que todas estas interpolaciones se podrán parecer más o menos a sus predecesoras. Si escogemos un punto que se encuentre en el centro de la matriz, obtendremos una melodía que compartirá las diferentes características de nuestras cuatro entradas en una misma proporción.

También se nos permite crear una transición entre los diferentes puntos de nuestra matriz de interpolaciones. Podemos tomar dos caminos:

- Arrastrar el cursor manualmente para ir recorriendo las diferentes melodías generadas, o incluso por los propios *inputs*. Aunque también podemos usar esta función para marcar un ritmo determinado sin necesidad de crear una transición entre varias.

- Dibujar un patrón de seguimiento. *Beat Blender* nos permite dibujar un recorrido por el que nuestro cursor pasará automáticamente sin necesidad de tocarlo. Esto nos puede servir para, por ejemplo, recorrer las cuatro esquinas de la matriz en un solo paso, yendo de una a otra por cada interpolación.

Aparte de estas funcionalidades, también se nos da la opción de cambiar los *bpm* en un rango de 1 a 200. De forma predeterminada contamos con 120 *bpm*, lo cual es una velocidad de pulsos muy adecuada para captar cada sonido que se reproduzca. No obstante, hemos de recordar que cada género musical suele venir acompañado de un ritmo específico. Así como la samba suele colocarse entre los 70 y 90 *bpm*, el *dance*, lo podemos catalogar normalmente como 128 *bpm*.

Por último, *Beat Blender* nos facilita una cómoda manera de guardar nuestros avances si nuestro deseo es recuperarlo en algún momento. Se nos facilitará un enlace personal que nos redigirá de nuevo a la página de inicio, pero con nuestras mezclas creadas desde un principio. Junto con el link, nos aparecerá la opción de compartir nuestras hazañas mediante *Facebook* o *Twitter*.

2.3 Latent Loops

Esta aplicación está creada por el departamento de *Pies's Shop* de *Google*. En cuanto a funcionalidad y aspecto es muy semejante a *Beat Blender*, salvo que *Latent Loops* cuenta con un amplio abanico de posibilidades a la hora de editar nuestros sonidos.

Por un lado, tenemos la opción de crear nuestras propias paletas musicales, al igual que en el ejemplo anterior, pero en este caso también podemos dibujar la entrada que queramos. De este modo se nos simplifica mucho más la creación de música.

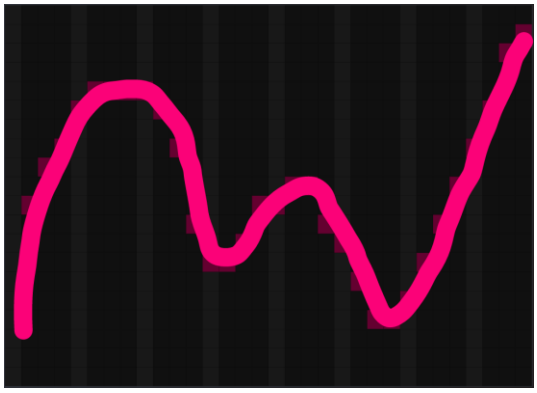

Figura 2-6: Dibujo de la melodía

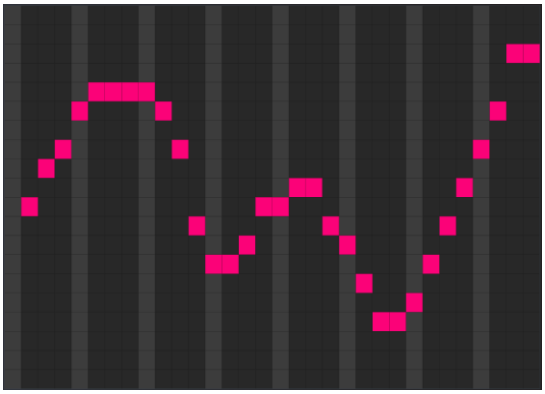

Figura 2-7: Transcripción del dibujo a melodía

Por otro lado, contamos con un menú de opciones lateral que nos ofrece un montón de posibilidades, como son:

- Tempo. Funciona del mismo modo que en aplicaciones anteriores, en un intervalo entre 80 y 280 *bpm*.

- Dimensión de la matriz de datos. Esta funcionalidad es muy interesante, ya que nos aporta construir más o menos interpolaciones acordes a nuestros criterios. El rango de tamaño de la matriz es desde un 4x4 a un 12x12. Cabe destacar que cuatro del total de pistas de audio serán siempre las entradas, por lo que generaremos N-4 interpolaciones, siendo N el tamaño de la matriz (N=100 si matriz=10x10).

- Utilizar salida MIDI. Se nos ofrece la opción de usar un dispositivo MIDI como salida, en el que podemos configurar la duración de cada nota en ms (de 10 a 1000), usar el reloj MIDI, o cambiar el tipo de instrumento, sobrentendiendo que este será un sonido digital.

- Instrumento web. En caso de descartar la opción anterior, podemos escoger entre un piano convencional o uno con sintetizador integrado.

- Escala. Un simple desplegable nos ofrece la posibilidad de contar con una escala mayor, menor o cromática.

- Armonía. En caso de que queramos una sonoridad de alguna nota en cuestión, basta con señalizarlo en el desplegable de escalas, donde veremos una escala completa desde do (C en notación musical) a si (B).

- Guardar. Idéntico a *Beat Blender*, guardamos nuestro progreso a modo de un link a la misma página, pero con nuestras canciones ya creadas.

- Descargar. Si ya has acabado con tu proyecto, ¿por qué no poder descargar los archivos generados? De clicar en este botón seremos capaces de obtener todo el conjunto de entradas e interpolaciones en un fichero comprimido zip.

Como podemos observar, *Latent Loops* nos ofrece un montón de posibilidades que enriquecen sin lugar a dudas el funcionamiento de esta aplicación. Pero incluso es capaz de guardar en su otro lateral una tabla a modo de registro, por si nos convenció alguna melodía generada y la queremos usar en futuros experimentos. E incluso es capaz de reproducir una melodía completa formada por pistas que hayamos seleccionado nosotros mismos.

2.4 Conclusiones del estado del arte

En este apartado se han expuesto y descrito una serie de aplicaciones enfocadas al mundo de la música. Cada una ofrece un modo más sencillo o complejo de crear interpolaciones de melodías. Estas tan sólo han sido un aperitivo de todo lo que se puede encontrar. A día de hoy están más enfocadas a dar ideas a aquellos que necesiten una pequeña ayuda, motivación o idea de qué o cómo pueden mejorar aquello que ya han creado. En un futuro esto puede llegar a desembocar en un manantial de música autogenerada que no requiera de supervisión alguna a la hora de determinar si, en relación con lo que se quiera llegar a conseguir, es un producto válido y cumple su objetivo. Pero ¿con esto se consigue frustrar las ideas de los grandes compositores del momento?

3 Definición del proyecto

En esta sección se comentan los objetivos principales del proyecto, así como las hipótesis que lanzaremos y que serán discutidas en el **apartado 5** de este documento. Trataremos de igual forma el alcance del proyecto y sus restricciones, que nos harán tomar unas u otras decisiones en función de los factores del proyecto.

3.1 Objetivos

A continuación, se presentan los objetivos del proyecto.

O1. Comprender el código implementado por Alberto Prudencio de Dueñas para una mayor comodidad a la hora modificar o adaptar el código, así como conocer el funcionamiento de este y el uso que le da a los diferentes sonidos de su biblioteca de *drums*.

O2. Realizar un exhaustivo trabajo de investigación para poder conocer de mejor forma las actuales aplicaciones que se están desarrollando.

O3. Estudiar las librerías más usadas por estas aplicaciones como *TensorFlow, MusicVAE, Tone* o *P5,* que nos facilitarán la creación de nuestro proyecto.

O4. Comprender que es una *LSTM* y cómo funciona.

O5. Definir un modelo que sea capaz de entrenar, mediante una *RNN* y una *LSTM*, toda una gama de melodías percutidas que nos servirán para interpolar las diferentes entradas que recibirá nuestra aplicación.

O6. Construir una transición a modo de puente entre la aplicación desarrollada por Alberto Prudencio de Dueñas y este proyecto.

O7. Calificar las salidas producidas mediante una función de evaluación.

3.2 Hipótesis

Habiendo dejado claros los objetivos, vamos a formular una serie de hipótesis que queremos demostrar en este proyecto.

H1. Las melodías interpoladas serán diferentes gracias al modelo entrenado con *RNN y LSTM*.

H2. La función de evaluación se adaptará correctamente a los cambios impuestos en la entrada. Es decir, si la entrada difiere en un par de variables frente a otra del conjunto de pruebas, esto apenas se verá reflejado en la calificación que saldrá como resultado de la función de evaluación, y, por lo contrario, si estas variables se ven muy afectadas se reflejará en el resultado final.

H3. Una melodía con mayor calificación se ajustará más a una que tenga una variedad de instrumentos sin exceso y con una ejecución controlada.

H4. No todas las clasificaciones serán buenas. Puede que clasifiquemos como mala una melodía que personalmente nos agrade o que por lo contrario el programa vea como bueno el "ruido".

3.3 Alcance

Definimos el alcance de nuestro proyecto como una meta que nos imponemos, un desarrollo completo de la aplicación y sus funcionalidades.

Este proyecto tendrá como alcance la construcción de un sistema capaz de interpretar melodías y desdibujarlas en ficheros que podamos tratar. Estos ficheros contendrán la información de cada nota de la melodía y sus características como veremos en el apartado 4.1. Dotar a la aplicación de la capacidad de interpolar entre dos melodías percutidas y clasificar las salidas generadas mediante una función de evaluación.

3.4 Restricciones

Como cualquier proyecto que se precie, debemos tener claras cuáles son nuestras capacidades. No sólo basta con establecerse un objetivo concreto, también es importante que somos capaces de conseguir con nuestros recursos. Estos recursos pueden ser tanto materiales como de tiempo o conocimientos.

R1. Durante el vigente curso he tenido que lidiar con que las asunciones que se hicieron a principio de curso sobre el uso de determinadas herramientas no fueron correctas, lo que hizo que se cambiara por completo la planificación del proyecto.

R2. De igual modo y durante el mismo momento, realizaba mi periodo de prácticas de empresa en el Instituto de Ingeniería del Conocimiento. Una vez finalicé las prácticas curriculares opté por continuar con extracurriculares. A día de hoy soy un contratado más en la empresa, por lo que mis responsabilidades han crecido exponencialmente este año. Esto hace que el tiempo de inversión en el proyecto se vea afectado de forma directa.

R3. El trabajo se reduce al tiempo que he estado en casa, pues el único medio con el que he contado ha sido un ordenador de sobremesa. Dispongo de un portátil, pero no consigue cumplir de forma satisfactoria cada una de las tareas en las que me he tenido que comprometer.

R4. Puesto que *TensorFlow* trabaja esencialmente con la *GPU*, ha hecho que la idea de trabajar fuera de casa se imposibilite completamente. Actualmente cuento con una *Nvidia GTX 1050*, suficiente para el uso que le queremos dar a nuestra aplicación.

R5. Partimos de los conocimientos básicos de *Machine Learning* estudiados en diversas asignaturas durante el curso en cuestión: Fundamentos de Aprendizaje Automático y Redes Neuronales. Esto ha hecho que el trabajo de investigación fuera superior al de desarrollo, ya que, para poder aplicar estos conocimientos en algo funcional, es necesario conocer de buena mano el funcionamiento de cada uno de los modelos y arquitecturas que queremos usar. Dentro de este estudio tenemos en cuenta librerías como *Tone.js, P5.js* y por supuesto *MusicVAE*, esencial en nuestro proyecto, así como los modelos de redes neuronales como *RNN* y *LSTM*.

4 Diseño

4.1 Diseño funcional

Para conocer bien el funcionamiento del proyecto es importante saber que iteraciones se hacen en él y que caminos puede llegar a tomar. Para ello, nos basaremos en el siguiente diagrama de flujo.

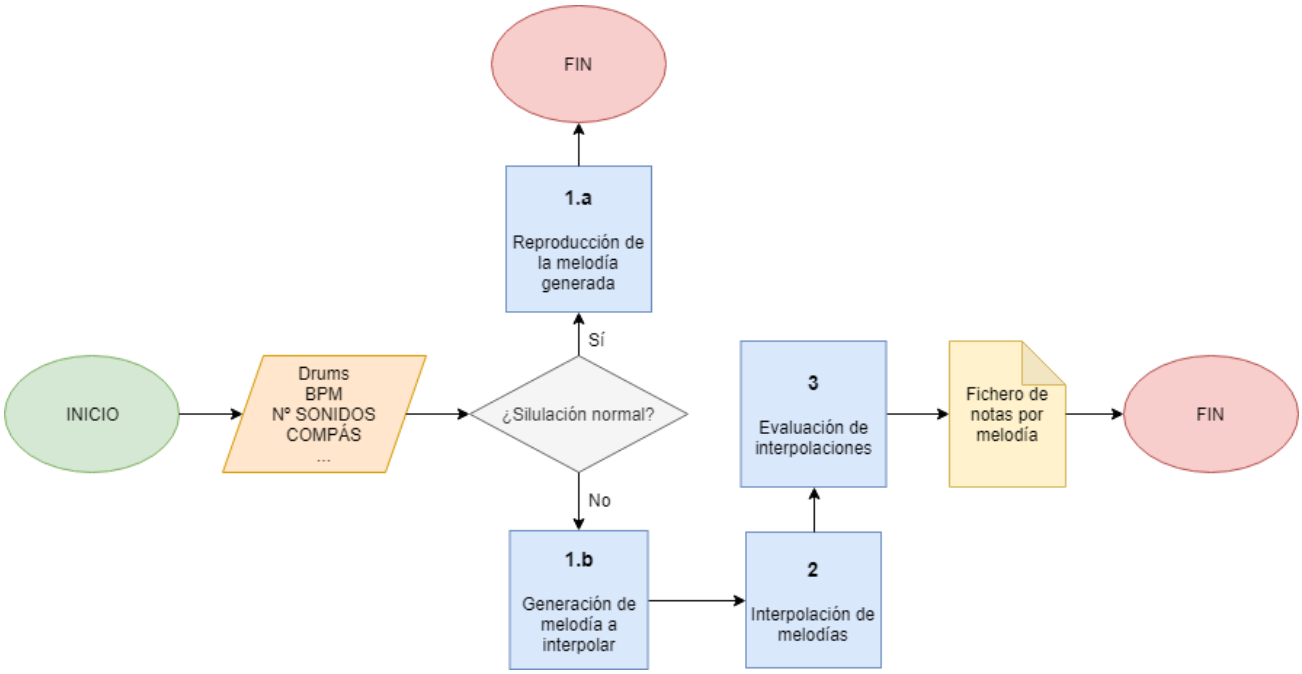

Diagrama 4-1: Diagrama de flujo de nivel 1 del proyecto

Como bien se define en el primer paso, se necesitan una serie de elementos para el correcto funcionamiento de la aplicación. Los elementos de entrada son los mismos para las dos formas de ejecución. Por un lado, si decidimos una simulación normal, ejecutaremos el programa creado por Alberto Prudencio de Dueñas, en el que se generará una melodía a partir de los argumentos de entrada **(1.a)**. Finalmente, se reproduce dicha melodía. Este es el caso base del que parte la implementación de este proyecto.

Si, por otro lado, se decide que no sea una simulación corriente, invocará al interpolador de melodías. Para ello, se generará un fichero legible para el interpolador. Este fichero cuenta con unas características concretas que el interpolador necesita para conocer qué reproducir y en qué momento reproducirlo.

Este fichero se genera a partir de las diferentes secuencias creadas por el *Artificial Composer*, extrayendo la información en dos ficheros distintos:

- **Fichero de mapeo de *samples*.** Este fichero contiene un mapeo de los sonidos seleccionados por el *Artificial Composer* con el tipo de sonido que es, a saber: *Claps, Cymbals, HiHatClosed, HiHatOpen, Kicks, Percussion* y *Snares*. El fichero de mapeo de *samples* será utilizado por la función de evaluación. En el **anexo A** se muestra un ejemplo del fichero.

- **Fichero de *preset-melodies*.** Este fichero es el que leerá el interpolador para saber cómo, qué y cuándo reproducir los *samples* previamente seleccionados. Para obtener la información necesaria se han de convertir los datos en el formato deseado. Por un lado, el interpolador de percusión está preparado para 9 sonidos distintos con un nombre específico, por lo que la primera tarea será obtener esos nombres para cada *sample*. Por otro lado, necesitamos el tempo de ejecución de cada sonido. Estos vienen dados por milisegundos, mientras que el interpolador lo que necesita es de que compás a que compás reproducir tal sonido.

  La obtención de este número se aplicará de la siguiente forma. Se sabe que en un compás de 4/4 hay, para 120 *bpm*, 4000 segundos. Si se quiere obtener cada cuantos milisegundos puede aparecer un sonido, se divide el total de milisegundos que puede durar la melodía (4000 en este caso) por el número de sonidos totales.

$$intervalo = \frac{milisegundos}{numSonidos}$$

  En el caso del ejemplo, se obtiene un intervalo de 125, es decir, cada 125 milisegundos se puede ejecutar un sonido (teniendo en cuenta que en el instante 0 también se puede reproducir). A continuación, se muestra una tabla con los intervalos y sus correspondientes tiempos.

Intervalo	Tempo	Intervalo	Tempo	Intervalo	Tempo	Intervalo	Tempo
0	0	1000	8	2000	16	3000	24
125	1	1125	9	2125	17	3125	25
250	2	1250	10	2250	18	3250	26
375	3	1375	11	2375	19	3375	27
500	4	1500	12	2500	20	3500	28
625	5	1625	13	2625	21	3625	29
750	6	1750	14	2750	22	3750	30
875	7	1875	15	2875	23	3875	31

Tabla 4-1: Equivalencia de intervalo y *tempo*

  En el **anexo B** se muestra un ejemplo de fichero de *preset-melodies*.

Para el siguiente paso, el interpolador se alimenta del fichero anteriormente mencionado y, con la ayuda de un par de selectores, permite colocar la melodía deseada a un lado u otro de las melodías a interpolar.

Una de las opciones a seleccionar, es la de *Generated*. Esta opción genera una melodía **(1.b)** a partir de un punto aleatorio del espacio latente creado por *Music VAE*.

La generación de esta melodía es posible gracias al entrenamiento del modelo desarrollado con *RNN* y *LSTM*. Este entrenamiento viene dado por el *storage* que proporciona *Google*. Existen varios ficheros que hacen que la generación del espacio latente a cargo de *Music VAE* sea posible. Tanto para el *encoder* como para el *decoder* tenemos varios ficheros que contienen la información del entrenamiento del modelo, así como las *bias*, el *kernel*, las funciones sigmoidales, los *inputs* y *outputs*.

Para generar la interpolación de melodías **(2)** basta con haber seleccionado un par de melodías como se ha descrito anteriormente, y arrastrar una de ellas a un lado o a otro. Al hacer esto, se irán generando interpolaciones a medida que se alejan las melodías del principio. Visualmente es más atractivo gracias a la implementación de una interfaz que crea un degradado de colores a la vez que pinta los diferentes sonidos en el cuadro que se ha generado.

Una vez se han generado las interpolaciones que se deseen, es hora de darle al *play* y disfrutar de una transición de melodía a melodía pasando por cada una de las interpolaciones creadas. Cuando este proceso acabe, se retornará al menú principal, sin perder en ningún momento los avances realizados. Una vez termina la ejecución del programa, se hace una petición a una *API* de *Python Flask*, la cual recibe en el cuerpo de la llamada un *array* con todos los sonidos reproducidos y los tiempos de ejecución. Será aquí donde se ejecute la función de evaluación **(3)** la cual será comentada en el **apartado 4.2.3**.

Finalmente se genera un fichero con las notas adjudicadas a cada melodía generada. Este contiene una fila por cada melodía calificada, en la que se encuentran los atributos valorados seguidos de una calificación final.

4.2 Descripción de modelos

Más allá del carácter visual y sonoro del que cuenta el proyecto, hay un amplio desarrollo y exhaustivo estudio de los modelos en los que se basa la aplicación. Vamos a hablar ampliamente de ellos y tratar sus ventajas frente a otros modelos al igual que sus inconvenientes.

4.2.1 Varitional Autoencoder

Para comprender el *VAE*, primero comenzaremos explicando el funcionamiento de una red simple e iremos incrementando paso a paso mayor complejidad a la red.

Una forma común de describir una red neuronal es una aproximación de alguna función que deseamos modelar. Sin embargo, también pueden considerarse como una estructura de daros que contiene información.

Supongamos que partimos de una red compuesta por unas cuantas capas de deconvolución [9]. Establecemos que la entrada sea siempre un vector inicializado con todos sus valores a 1. Luego, podemos entrenar la red para conseguir reducir el error cuadrático medio entre este vector y una imagen destino.

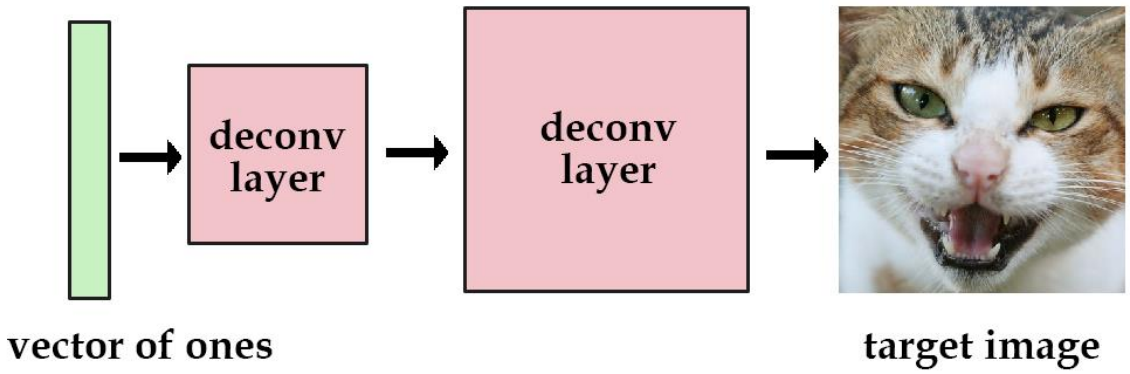

Figura 4-1: Vector de 1s a imagen

Ahora, vamos a intentar codificar múltiples imágenes, y en lugar del vector anterior, usaremos un vector *one-hot* [10] para la entrada. Supongamos que el vector [1, 0, 0, 0] equivale a un gato y el [0, 1, 0, 0] a un perro. Parece que funciona, pero sólo seríamos capaces de almacenar un par más de imágenes. La solución parece ser sencilla. Un vector con una dimensión mayor para clasificar un número infinitamente superior.

Para solucionar este problema, vamos a trabajar con un vector compuesto de número reales en lugar de un vector *one-hot*. Daremos unos valores de, por ejemplo, [3.3, 4.5, 2.1, 9.8] para la imagen del gato y [3.4, 2.1, 6.7, 4.2] para la del perro. A estos vectores iniciales los conoceremos como nuestras variables latentes.

Como vimos en el apartado 2.2.3, un *autoencoder* agrega un nuevo componente que trata la imagen original y la codifica en vectores como los recientemente comentados. Es entonces cuando las capas de deconvolución decodifican los vectores para reconstruir la imagen original a partir de dicho vector.

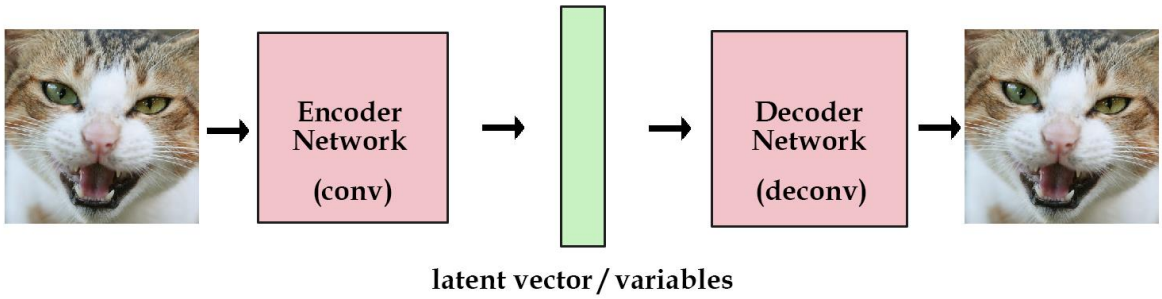

Figura 4-2: Modelo de autoencoder sobre el que construir el *VAE*

Actualmente, el modelo que hemos construido cuenta con un uso práctico, idéntico al de un *autoencoder* convencional. Sin embargo, estamos tratando de construir un modelo generativo, no sólo reconstruir imágenes. Aún no podemos generar nada, ya que no sabemos cómo crear vectores latentes que no hayan sido codificados a partir de una imagen.

Vamos a agregar una restricción a la red de codificación, que obligue a la red a generar vectores latentes que sigan una distribución gaussiana. Esta es la restricción que consigue separar la definición de un *autoencoder* de un *VAE*. Ahora es más sencillo generar imágenes nuevas. Lo único que necesitamos es muestrear un vector latente de la unidad gaussiana y pasárselo al decodificador para que este pueda generar una imagen distinta a las de la entrada.

Para definir la relación de precisión con la del error en la ejecución, hay que tener en cuenta las dos pérdidas que se nos dan: la pérdida generativa, asociada a un error cuadrático medio que mide la precisión con la que la red ha sido capaz de reconstruir la imagen, y una pérdida latente, que es la divergencia KL [11] que mide cuanto de cerca están las variables latentes de la unidad gaussiana.

```
generation_loss = mean(square(generated_image - real_image))
latent_loss = KL-Divergence(latent_variable, unit_gaussian)
loss = generation_loss + latent_loss
```

Para optimizar la divergencia de KL, necesitamos que en lugar de que el codificador genere un vector a valores reales, genere un vector de medias y un vector de desviaciones típicas.

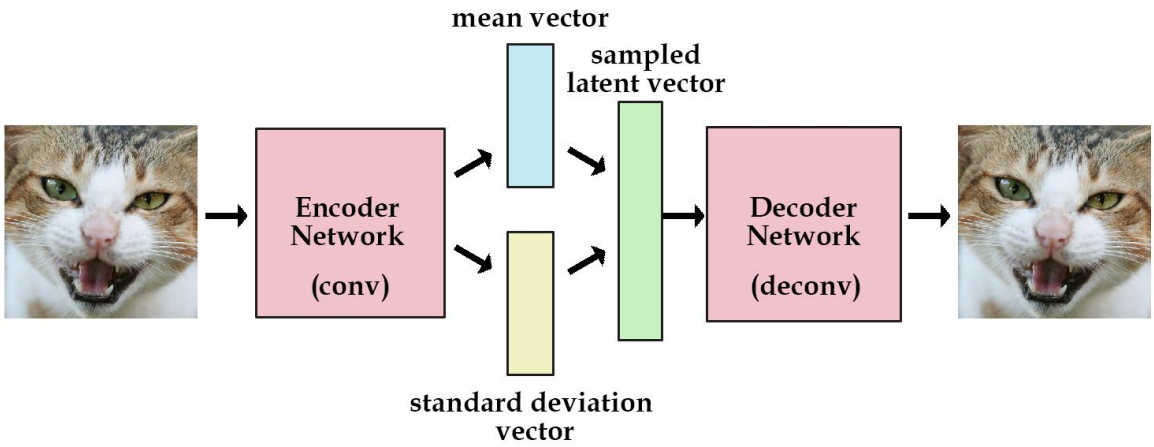

Figura 4-3: Arquitectura de un modelo *VAE*

Esto nos permite calcular la divergencia de KL de la siguiente manera:

```
# z_mean and z_stddev son dos vectores generados por el encoder
latent_loss = 0.5 * tf.reduce_sum(tf.square(z_mean) + tf.square(z_stddev) -
tf.log(tf.square(z_stddev)) - 1,1)
```

Cuando calculamos la pérdida de la red del decodificador, podemos muestrear las desviaciones típicas y agregar la media para usarla como nuestro vector latente.

```
samples = tf.random_normal([batchsize,n_z],0,1,dtype=tf.float32)
sampled_z = z_mean + (z_stddev * samples)
```

Además de permitirnos generar variables latentes aleatorias, también hace que la generación de resultados de nuestra red mejore considerablemente.

Esta restricción obliga al codificador a ser muy eficiente creando variables latentes en abundancia de información. Esto hace que la generación de nuevas salidas mejore, por lo que las variables latentes que generemos aleatoriamente o que obtuvimos de la codificación de imágenes sin entrenamiento, puedan llegar a producir un mejor resultado cuando se descodifiquen.

4.2.2 Long short-term memory (LSTM)

Las redes de memoria a largo plazo, generalmente llamadas *LSTM* [12], son un tipo especial de *RNN*, capaz de aprender dependencias a largo plazo. Fueron introducidas por Hochreiter y Schmidhuber (1997). Actualmente están en auge ya que funcionan francamente bien en una amplia variedad de problemas.

Las *LSTM* están diseñadas explícitamente para evitar el problema de dependencia a largo plazo. Recordar información durante largos períodos de tiempo es prácticamente su comportamiento predeterminado. Todas las redes neuronales recurrentes tienen la forma de una cadena de módulos repetitivos de la red neuronal. En las *RNN* estándar, este módulo de repetición contará con una estructura muy simple, con una sola capa de *tanh* [13].

Las *LSTM* también tienen esta estructura de cadena, pero el módulo de repetición tiene una estructura diferente. En lugar de tener una sola capa de red neuronal, hay cuatro que interactúan de una forma muy especial.

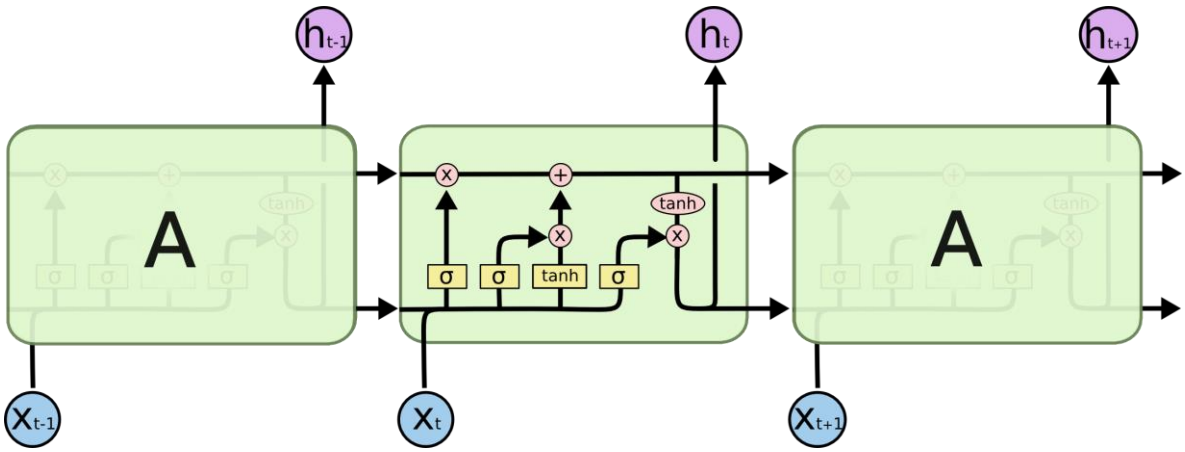

Figura 4-4: Módulo de repetición en una *LSTM*

En la **Figura 3-4** cada línea transporta un vector completo, desde la salida de un nodo hasta las entradas de otros. Los círculos del interior representan operaciones puntuales, como la adición de vectores, mientras que los cuadros amarillos son capas de la red neuronal ya aprendidas. Las líneas que se fusionan denotan la concatenación, mientras que una línea de bifurcación denota el contenido que se está copiando y las copias que van a diferentes ubicaciones.

La principal clave para este tipo de redes es el estado de la celda, la línea horizontal que recorre la parte superior del diagrama. El estado de la celda tiene un funcionamiento semejante al de una cinta transportadora. Atraviesa la cadena con ligeras interacciones, aunque es muy común que no sufra cambios durante el proceso.

Estas redes tienen la capacidad de eliminar o agregar información al estado de la celda, cuidadosamente regulado por estructuras llamadas *gates*. Las *gates* son una forma de permitir que la información avance. Se componen de una red neuronal con una función de activación sigmoidal y una operación de multiplicación puntual.

La capa de la sigmoide produce un resultado entre 0 y 1 que describen la cantidad de cada componente que debe dejarse pasar. Un valor de cero significa que el valor no continuará, mientas que un valor de uno hace que se transmita la totalidad de la información. Una *LSTM* cuenta con tres de estas *gates* para proteger y controlar el estado de la celda.

El primer paso en nuestro *LSTM* es decidir qué información vamos a eliminar del estado de la celda. Esta decisión es tomada por una capa sigmoidal llamada *forget gate layer*. Podemos observar que en h_{t-1} y x_t, y en las salidas que se genera un número entre 0 y 1 para cada número en el estado de la celda C_{t-1}. El valor 1 representa mantener por completo el estado actual mientras que un 0 significa deshacerse de todo.

El siguiente paso es decidir qué información nueva vamos a almacenar en el estado de la celda. Esto tiene dos partes:

1. Una capa sigmoidal llamada *input gate layer* que decide qué valores actualizaremos.

2. Una capa *tanh* que cree un vector de nuevos valores candidatos, C_t, que podría agregarse al estado.

3. En el siguiente paso, combinaremos estos dos para crear una actualización para el estado.

Ahora es el momento de actualizar el estado de la celda anterior, C_{t-1}, en el nuevo estado de la celda C_t. Multiplicamos el estado antiguo por f_t olvidando las cosas que decidimos despreciar antes. Entonces añadimos $i_t * C_t$. Estos son los nuevos valores candidatos, en función de cuánto decidimos actualizar cada valor de estado.

Finalmente, tenemos que decidir qué vamos a producir. Esta salida se basará en el estado de nuestra celda, pero será una versión filtrada. Primero, ejecutamos una capa sigmoidal que decide qué partes del estado de la celda vamos a generar. Entonces, ponemos el estado de la celda a través de la capa *tanh*, para forzar a que los valores sean entre -1 y 1. Y multiplicarlo por la salida de la *gate* sigmoidal, de modo que sólo emitimos las partes que nosotros decidamos.

4.2.3 Función de evaluación

Para evaluar las distintas melodías generadas se ha optado por una serie de patrones que describiremos a continuación. Cada uno de ellos cuenta con un valor ponderado mientras que la suma total de ellos es de 10. Si se quiere obtener un resultado fiable, se deberá dar mayor importancia a aquellos criterios que se darían en un entorno real, al igual que se menospreciará aquello que no sea relevante.

A continuación, se definen los criterios que se han tenido en cuenta para la creación de la función de evaluación.

1. El primer sistema de evaluación se evalúa sobre 6 puntos, ya que es el que cuenta con los criterios fundamentales para que melódicamente suene bien. Comprueba aspectos básicos de la composición de *drums*, como son la variedad de sonidos siempre y cuando no haya una saturación de los mismos, es decir, que exista un cierto equilibrio. Para la comprobación de este primer sistema de evaluación se tendrán en cuenta los siguientes criterios:

    - El número de *Kicks* es distinto de 0.
    - El número de *Claps* o *Snares* es distinto de 0.
    - El número de *HiHatClosed* es distinto de 0.
    - El número de *HiHatOpen* es distinto de 0.
    - El número de sonidos de cada tipo no es superior a 2. Existen diversos sonidos para cada tipo, por lo que si la melodía está formada por 3 o más sonidos de un mismo tipo se penalizará.

    Si cada uno de estos criterios se cumple sumaremos 1 al total, hasta poder llegar, por lo tanto, a un total de 5 puntos. Si no se cumple el último caso, este penalizará con -0.125 sobre la nota calculada para el primer sistema de evaluación. Finalmente se pondera la nota sobre 6.

2. Este sistema de evaluación trata de darle peso a los sonidos que marcan el ritmo en una melodía. Por lo tanto, se dará mayor importancia a los *samples* de *Kicks* y *Claps*. Pero no solo con esto basta, ya que el ritmo no se marca en cualquier pulso, si no que se tratará de favorecer a estos que se ejecutan en un momento determinado del compás, y penalizar a los que no lo hacen.

    Se define un momento idóneo como los pulsos que marcan el ritmo en una melodía. En este caso, al tener un compás de 4/4 con un total de 32 disposiciones dentro del mismo, se optará por definir un buen pulso para aquellos que cumplan la siguiente condición:

```
if df_melodia['tempo'][linea]%4 ==0:
```

De este modo, nos aseguramos que estos pulsos sean los idóneos para la melodía. Se beneficiará a estos que cumplan la condición con un punto, hasta poder alcanzar un total de 8 en cada caso. De igual forma se penalizará los que no lo cumplan con un -0.125 sobre la nota del sistema de evaluación parcial. Finalmente, esta nota se ponderará sobre 3.

3. Para el último caso, se evaluará si la melodía contiene vocales en su estructura. Es importante que una melodía tenga ritmo, pero si esta viene acompañada de una parte vocal, su atracción puede mejorar. No obstante, este es un arma de doble filo, ya que, al igual que en el caso anterior, no podemos tener este tipo de sonidos en cualquier momento. Ya que se dispone de sonidos vocales que encajan perfectamente en la estructura definida (4 segundos de duración por sonido), si este se ejecuta en el primer instante, se sumará 1 al resultado de la evaluación parcial, siempre y cuando no aparezca otro por medio que estropee la sonoridad de la melodía. Este último caso y el que tan sólo aparezca un sonido vocal en un pulso erróneo, se penalizará con -0.5 puntos sobre dicha evaluación. Esta evaluación parcial cuenta con 1 punto posible sobre el total de la nota.

Ya que todas las notas son ponderadas en el tiempo en el que se calculan, basta con sumarlas y obtener la nota global de cada una de las melodías reproducidas.

$$NotaGlobal = notaEquilibrio + notaPulsos + notaVocal$$

El resultado final quedará registrado en un fichero de evaluación formado por un identificador de la melodía evaluada, las notas parciales separadas por *";"* y finalmente la nota final.

5 Implementación

En este apartado se define para cada uno de los procesos instanciados en el Diagrama 4-1, los ficheros, almacén de datos, librerías, tecnologías usadas y versión de las mismas.

A continuación, se define cada uno de los puntos desarrollados en el proyecto.

(1.a) El entorno de trabajo en el que se ha implementado este proceso ha sido *Eclipse* en su versión 4.11 para desarrollar la generación de ficheros de los que se alimenta el interpolador en *Java*. La librería *TinySound* para la simulación de melodías, apoyada en el almacén de datos formado por 158 archivos *.wav*, que constituyen cada uno de los *samples* usados tanto para la generación de melodías en este paso, como en el interpolador.

(1.b) Para la generación de música a partir del espacio latente, se ha usado la herramienta *Visual Studio Code* en su versión 1.35, muy cómoda y ágil para este tipo de aplicaciones que siguen un patrón de *npm* (versión 6.9.0) y *Node.js* (versión 10.16.0) que cuenta con una consola propia donde podemos instalar dependencias o lanzar nuestra aplicación web. La librería que hace que esto sea posible es *MusicVAE* con su última versión 1.1.5, ya que actualmente ha migrado a la librería *Music* [14] (1.8.0) de *Magenta*.

(2) La integración de este proceso sigue de la mano de *Visual Studio Code*, del que ya hemos hablado, pero cuenta con un par de librerías fundamentales para el uso de la aplicación. Por un lado, tenemos *P5.js* en su versión 0.8.0 la cual nos proporciona todo el visualizado que tiene la interfaz, los movimientos de melodías, las transiciones entre ellas, los degradados de melodía en melodía o simplemente el detallado de la web. Por otro lado, contamos con la librería *tone.js* en su versión 13.4.9, que nos aporta la interacción entre melodías, instrumentos y diferentes tonos. Gracias a ella se puede obtener la información relevante para la generación de ficheros de evaluación, ya que nos marca en cada momento que pista de audio se está escuchando al igual que sus tiempos de ejecución.

No hay que olvidar que la aplicación sigue una jerarquía de modelo vista controlador, en la que contamos con un único fichero *html* y *css*, al igual que un único archivo *javascript* que se apoya de las librerías ya mencionadas.

(3) La evaluación de interpolaciones corre a cargo de *Python* 3.7.14. *Python* permite ser todo lo ágiles que, con *javascript*, no se puede. Esto incluye sobre todo a lo que a los ficheros se refiere, ya que *javascript* no es un lenguaje para ello. Con un par de librerías tan útiles como son *numpy* [15] (versión 1.14.5) y *pandas* [16] (versión 0.24.2), se puede sentir la libertad y comodidad de tratar ficheros *txt* como si de un *csv* se tratara gracias a los *dataframes* que implementa.

Para la comunicación entre la web y la *API* se ha usado la librería *Flask* [17] en su versión 1.0.2. *Flask* es un *framework* escrito en *Python* y concebido para facilitar el desarrollo de aplicaciones web bajo el patrón *MVC* [18]. Con él, la comunicación entre ambas plataformas es tan sencilla como de a una llamada a una función se tratase. Para comunicar la web con la *API* también se ha usado *Ajax*.

Para el lanzamiento de la *API* basta con inicializar la dirección y su puerto. En este caso se ha seleccionado el *localhost* y el puerto 9000. El puerto de transferencia de datos de la web es el 3000.

Cabe destacar el uso en todo momento de la plataforma *GitLab*, donde se ha ido subiendo cada actualización del proyecto y donde se ha seguido un diagrama de tareas, etiquetadas según su vinculación en ese momento del proyecto: *created*, *ToDo*, *doing* y *done.*

6 Experimentos

En este apartado se definirán los distintos experimentos realizados y sus resultados finales. Primero se probará el funcionamiento de la librería base *MusicVAE*, y se comprobará si realmente marca la diferencia un *VAE* frente a un *autoencoder* convencional. A continuación, se probarán distintos ficheros de entrenamiento para varios sistemas de percusión y uno para melodía.

6.1 VAE frente a autoencoder

Para comprobar el fiel funcionamiento de la librería *MusicVAE* realizaremos una prueba ejecutando la aplicación por el camino 1.b definido en el **Diagrama 4-1**, con un *VAE* y un *autoencoder* convencional por separado.

Si se ejecuta el experimento sin lo que aporta un *VAE*: un vector de medias y un vector de desviaciones típicas, las melodías que se generen contarán, en el mejor de los casos, con una sonoridad adecuada. No obstante, no existirá transición alguna entre las dos melodías preseleccionadas.

En el siguiente ejemplo se muestra una ejecución sin usar la librería *MusicVAE*.

Figura 6-1: Ejecución de un *autoencoder*

Mientras que los segmentos de inicio (rojo) y final (morado) coinciden perfectamente con las secuencias originales (negro), los intermedios no son realistas. El espacio de salida tiene expresión, pero carece de realismo y suavidad.

Mientras que, si se ejecuta la aplicación con el VAE, se observa que existe una transición entre ambas melodías previamente seleccionadas.

Figura 6-2: Ejecución de un *VAE*

Observamos que las secuencias intermedias ahora si producen una transición suave. Las secuencias intermedias tampoco están restringidas a la selección de las notas iniciales. La selección de notas tiene más sentido musical en el contexto de los puntos seleccionados. En este ejemplo se satisfacen plenamente las propiedades de expresión, realismo y suavidad expuestas en el **apartado 2.1.2**.

6.2 Diferentes entrenamientos

Para asegurar una buena elección dentro de estos ficheros, se debe probar cada uno de los mismos y evaluar su composición en cuanto al cumplimiento de las tres reglas que se deben satisfacer.

Primero se probará un modelo entrenado para melodías cuyos sonidos pertenecen a escalas musicales, algo que no tiene nada que ver con la percusión que queremos acabar generando. A continuación, se probarán diferentes modelos para percusión. Dentro de las restricciones de estos modelos se encuentran diferencias dentro de la prioridad de cada uno: si está hecho para reconstrucciones, interpolaciones, o simplemente para *samplear*.

El problema que causa el primer modelo es la imposibilidad de ejecutar varios sonidos en un mismo pulso. Independientemente de esto, el modelo está entrenado para seguir una fluidez en cuanto a las notas se refiere, por lo que el modelo interpreta que, si tenemos una nota baja y a continuación una inmediatamente más alta, estamos en una subida de la escala y viceversa. Esto perjudica al objetivo de generar percusión, ya que cada sonido es independiente del otro.

Para el resto de modelos probados, se encuentras diferencias en cuanto a las notas globales obtenidas en cada uno de los modelos. Finalmente, se eligió aquel que mejores resultados daba. A continuación, se muestra una tabla con un ejemplo del modelo elegido frente a uno que se descartó por su bajo rendimiento. Para la ejecución de este experimento, se ha obviado el caso en el que aparece una vocal dentro de la melodía para facilitar el cálculo y ver claramente las diferencias.

ID	NotaGlobal
1	7.425
2	6.425
3	5.8375
4	5.425
5	4.425
6	5.05
7	5.65
8	5.65
9	4.85
10	5.85
11	5.425
12	4.45
13	4.2375
14	4.45
15	4.85
16	6.0375

Tabla 6-1: Evaluación de uno de los modelos no seleccionados

ID	NotaGlobal
1	7.425
2	7.425
3	7.425
4	7.425
5	6.8375
6	6.05
7	6.65
8	7.2375
9	7.05
10	6.85
11	6.85
12	5.45
13	5.45
14	5.45
15	5.85
16	6.0375

Tabla 6-2: Evaluación del modelo final

La nota del primer y último ID coinciden debido a que es la nota global adjudicada a las melodías preseleccionadas, y que no varían en ninguna de las dos ejecuciones. Esto se ha realizado de esta forma para que ningún modelo partiera con desventaja, ejecutando el mismo ejemplo para ambos.

7 Conclusiones y trabajo futuro

Una vez finalizado el proyecto y habiendo experimentado con él, llega la hora de concluir y comprobar si los objetivos se han cumplido y las hipótesis que se formularon han sido resueltas satisfactoriamente.

7.1 Conclusiones

A continuación, se enumeran cada uno de los objetivos propuestos y las hipótesis que hemos querido corroborar.

O1. Se ha comprendido en su totalidad el funcionamiento del *Artificial Composer*, así como el uso que le da a la librería de sonidos y a la biblioteca de *samples* proporcionada. Las dificultades han sido fácilmente solventadas gracias a las explicaciones de su propio autor.

O2. Se ha realizado un exhaustivo trabajo de investigación en el que se ha comprendido el complejo funcionamiento de las diferentes aplicaciones que actualmente se están desarrollando.

O3. La comprensión de las librerías usadas en la aplicación ha sido una de las tareas más complicadas y que más trabajo han costado. De igual manera, se ha resuelto satisfactoriamente.

O4. Se ha comprendido el funcionamiento de las *LSTM* y el uso que se le puede dar teniendo en cuenta las necesidades de cada proyecto.

O5. Se ha integrado un modelo basado en *RNN* y *LSTM* para poder crear interpolaciones a partir de las diferentes entradas suministradas.

O6. La imperiosa necesidad de construir un puente entre ambas aplicaciones se ha cumplimentado de forma totalmente correcta. Desde un principio no se pudo definir dicha estructura debido a la alta dependencia con otros objetivos anteriormente descritos, como son la comprensión del código del *Artificial Composer*, o la de las librerías a utilizar.

O7. Se ha conseguido crear una función de evaluación que sigue unos criterios previamente definidos.

H1. Gracias al experimento desarrollado en el apartado **6.1** podemos concretar que sí, las melodías generadas mediante la interpolación son diferentes a las que no usan un modelo entrenado con *RNN y LSTM*. Es más, se obtienen mejores resultados aplicando este tipo de redes, como se puede ver en la **Figuras 6-1** y **Figura 6-2**.

H2. Se ha comprobado que dos melodías aparentemente diferentes pueden llegar a obtener resultados iguales. Esto se debe a que se siguen los mismos criterios para todas las melodías y puede ser que donde sume una la otra no lo haga y viceversa. También se ha dado el caso de melodías similares en cuanto a ritmo que carecían de similitud en los *samples* usados, pero, al pertenecer a la misma categoría, se han obtenido calificaciones similares.

H3. Gracias a las condiciones definidas en la función de evaluación, se puede confirmar que a mayor calificación existe un mayor equilibrio y control sobre la ejecución de la melodía.

H4. Se ha dado el caso de melodías que apenas tenían sonidos y que han obtenido mejor calificación que algunas que aparentemente sonaban mejor.

Involucrarse en un proyecto de esta talla ha sido un reto. Un reto que yo elegí y que me ha motivado a seguir ampliando conocimientos tanto de los modelos a usar en la generación de música, como a seguir practicando a mi forma más modesta. Por lo general ha sido un proyecto satisfactorio.

Se han ampliado considerablemente los conocimientos de lo que partía al comienzo del proyecto, a la par que he podido comprobar de primera mano los resultados que surgían al avanzar en él.

7.2 Trabajo futuro

Como trabajo futuro se exponen una serie de ideas que surgieron durante el desarrollo de este proyecto y que pueden servir, de cara a una ampliación del mismo, para incrementar el alcance y romper los límites de la música en el mundo de la informática.

- Capacidad de construcción de una canción completa a partir de las interpolaciones generadas.
- Implementación de diferentes instrumentos que pueden acompañar conjuntamente a la percusión ya generada.
- Sistema de configuración y modulación del sonido en tiempo real.
- Plataforma de *streaming* capaz de ir mezclando los diferentes inputs en tiempo real que los usuarios vayan suministrando.
- *Bot* de Twitter capaz de subir enlaces a nuestras canciones generadas favoritas.
- Función de evaluación basada en algoritmos más complejos.

Referencias

[1] MCCARTHY, Lauren; REAS, Casey; FRY, Ben. Getting Started with P5. js: Making Interactive Graphics in JavaScript and Processing. Maker Media, Inc., 2015.

[2] MANN, Yotam. Música interactiva con tono. js En los procedimientos de la 1 ª Conferencia anual de audio web . 2015.

[3] ABADI, Martín, et al. Tensorflow: un sistema para el aprendizaje automático a gran escala. En 12th {USENIX} Simposio sobre diseño e implementación de sistemas operativos ({OSDI} 16) . 2016. p. 265-283.

[4] ROBERTS, Adam, et al. A hierarchical latent vector model for learning long-term structure in music. arXiv preprint arXiv:1803.05428, 2018.

[5] CASELLA, Pietro; Paiva, ana. Magenta: Una arquitectura para la composición automática en tiempo real de la música de fondo. En Taller Internacional de Agentes Virtuales Inteligentes . Springer, Berlín, Heidelberg, 2001. p. 224-232.

[6] HA, David; ECK, Douglas. A neural representation of sketch drawings. arXiv preprint arXiv:1704.03477, 2017.

[7] ENGEL, Jesse; HOFFMAN, Matthew; ROBERTS, Adam. Latent constraints: Learning to generate conditionally from unconditional generative models. arXiv preprint arXiv:1711.05772, 2017.

[8] ENGEL, Jesse, et al. Neural audio synthesis of musical notes with wavenet autoencoders. En Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017. p. 1068-1077.

[9] NOH, Hyeonwoo; HONG, Seunghoon; HAN, Bohyung. Learning deconvolution network for semantic segmentation. En Proceedings of the IEEE international conference on computer vision. 2015. p. 1520-1528.

[10] BUCKMAN, Jacob, et al. Thermometer encoding: One hot way to resist adversarial examples. 2018.

[11] YU, Dong, et al. KL-divergence regularized deep neural network adaptation for improved large vocabulary speech recognition. En 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 2013. p. 7893-7897.

[12] HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. Neural computation, 1997, vol. 9, no 8, p. 1735-1780.

[13] KALMAN, Barry L.; KWASNY, Stan C. Why tanh: choosing a sigmoidal function. En [Proceedings 1992] IJCNN International Joint Conference on Neural Networks. IEEE, 1992. p. 578-581.

[14] ROBERTS, Adam; HAWTHORNE, Curtis; SIMON, Ian. Magenta. js: A javascript api for augmenting creativity with deep learning. 2018.

[15] VAN DER WALT, Stefan; COLBERT, S. Chris; VAROQUAUX, Gael. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 2011, vol. 13, no 2, p. 22.

[16] MCKINNEY, Wes. pandas: a foundational Python library for data analysis and statistics. Python for High Performance and Scientific Computing, 2011, vol. 14.

[17] GRINBERG, Miguel. Flask web development: developing web applications with python. " O'Reilly Media, Inc.", 2018.

[18] POP, Dragos-Paul; ALTAR, Adam. Designing an MVC model for rapid web application development. Procedia Engineering, 2014, vol. 69, p. 1172-1179.

Glosario

API	Application Programming Interface
Autoencoder	Redes neuronales cuyo objetivo es el de generar nuevos datos a partir de una compression de la entrada en un espacio latente y luego reconstruyendo la salida en base a una información previamente suministrada
Encoder	Crea un espacio latente con los datos a aprender
Decoder	Decodifica, descomprime o reproduce el mismo ejemplo del vector
Sample	Acto de tomar un trozo de un sonido previamente grabado para darle un nuevo uso como un instrumento musical

Anexos

A Fichero de mapeo de samples

```
50    0     Kicks
36    2     HiHatsClosed
38    2     Cymbals
51    2     Claps
36    3     HiHatsClosed
42    4     Cymbals
36    5     HiHatsClosed
46    8     Kicks
45    10    Kicks
36    16    HiHatsClosed
51    18    Claps
42    20    Cymbals
36    22    HiHatsClosed
45    22    Kicks
36    25    HiHatsClosed
50    26    Kicks
38    30    Cymbals
46    30    Kicks
```

B Fichero de preset-melodies

```
var presetMelodies = {
'PruebaDrumX' : {
      notes: [
            {pitch: 50, quantizedStartStep: 0, quantizedEndStep: 1},
            {pitch: 36, quantizedStartStep: 2, quantizedEndStep: 3},
            {pitch: 38, quantizedStartStep: 2, quantizedEndStep: 3},
            {pitch: 51, quantizedStartStep: 2, quantizedEndStep: 3},
            {pitch: 36, quantizedStartStep: 3, quantizedEndStep: 4},
            {pitch: 42, quantizedStartStep: 4, quantizedEndStep: 5},
            {pitch: 36, quantizedStartStep: 5, quantizedEndStep: 6},
            {pitch: 46, quantizedStartStep: 8, quantizedEndStep: 9},
            {pitch: 45, quantizedStartStep: 10, quantizedEndStep: 11},
            {pitch: 36, quantizedStartStep: 16, quantizedEndStep: 17},
            {pitch: 51, quantizedStartStep: 18, quantizedEndStep: 19},
            {pitch: 42, quantizedStartStep: 20, quantizedEndStep: 21},
            {pitch: 36, quantizedStartStep: 22, quantizedEndStep: 23},
            {pitch: 45, quantizedStartStep: 22, quantizedEndStep: 23},
            {pitch: 36, quantizedStartStep: 25, quantizedEndStep: 26},
            {pitch: 50, quantizedStartStep: 26, quantizedEndStep: 27},
            {pitch: 38, quantizedStartStep: 30, quantizedEndStep: 31},
            {pitch: 46, quantizedStartStep: 30, quantizedEndStep: 31},

      ],
      color : [113, 20, 221]
},};
```